Speeding up software with VecCore

G Amadio[1], P Canal[2], D Piparo[1], S Wenzel[1]
[1]CERN, [2]FNAL

E-mail: amadio@cern.ch

Abstract. Portable and efficient vectorization is a significant challenge in large software projects such as GeantV, ROOT, and experiments' frameworks. Nevertheless, fully exploiting SIMD parallelism will be a required step in order to bridge the widening gap between the needs and availability of computing resouces for data analysis and processing in particle physics. Although there are SIMD libraries that wrap compiler intrinsics into a convenient interface, they do not always support all available architectures, or they only perform well in some of them. The VecCore library was created to address some of these performance and portability issues by providing a unified abstraction layer on top of existing libraries, such as Vc or UME::SIMD. In this article, we discuss VecCore's programming model for SIMD code and some use cases in HEP software packages such as VecGeom and GeantV.

1. Introduction

Experiments at CERN have an ever increasing demand for computational resources, be it for simulated collisions or data analysis. In the last few decades, this demand has been met by hardware upgrades of the Worldwide LHC Computing Grid. However, with the next runs of the LHC approaching, the expected increases in beam luminosity will push this demand far beyond the limits of what further hardware upgrades can reach. Therefore, in order to bridge the widening gap between the needs of the HEP community and the existing computing resources, HEP software will need to be optimized to be able to fully exploit SIMD and multithreading parallelism available in modern hardware.

One of the key areas through which performance can be substantially improved in HEP software is SIMD vectorization. Even so, writing efficient SIMD vectorized code is a significant challenge in many large software projects such as GeantV[1, 2], ROOT[3], and the experiments' frameworks. On the one hand, compilers cannot reliably auto-vectorize simulation and analysis software. One the other hand, while there are SIMD libraries that wrap the cumbersome compiler intrinsics functions into a more convenient interface, these libraries often target specific architectures, which means they may not work on ARM, for example. Therefore, it makes sense to be able to switch between libraries depending on the target architecture, to be able to use the best option on each platform.

2. The VecCore Library

The VecCore library was created in order to solve the lack of portability and unreliable performance problems usually associated with SIMD code. VecCore provides a simple API for users to express their SIMD-enabled algorithms that can be dispatched to different backend implementations, such as the SIMD libraries Vc [4] and UME::SIMD [5], or even CUDA, if the code has the proper

annotations. Vc in general has very good performance on systems that support the AVX2 SIMD instruction set, but its latest release, Vc 1.3.3, has no support for the more recent AVX512. Conversely, UME::SIMD focuses on scalar emulation for multiple architectures, while optimizing newer architectures first, which makes it perform quite well on AVX512, but the performance double precision types on AVX2 may not be as good as Vc[6]. In this paper we use the latest release, UME::SIMD 0.8.1.

Using VecCore, developers can write generic computational kernels using abstract types that map to the different concrete types in each backend. The API for vectorization provided by VecCore is architecture-agnostic, making it easy to fallback to scalar types on platforms that do not have SIMD instructions. Moreover, users do not have to change their code to switch between backends or target architecture. The API covers the essential parts of the SIMD instruction set that allows one to write many of the numerical algorithms needed by HEP software. The API functions are shown in Listing 1 and their operations are illustrated in Figure 2.

```
namespace vecCore {

template <typename T> struct TypeTraits;

template <typename T> using Mask   = typename TypeTraits<T>::MaskType;
template <typename T> using Index  = typename TypeTraits<T>::IndexType;
template <typename T> using Scalar = typename TypeTraits<T>::ScalarType;

// Vector Size
template <typename T> constexpr size_t VectorSize();

// Get/Set
template <typename T> Scalar<T> Get(const T &v, size_t i);
template <typename T> void Set(T &v, size_t i, Scalar<T> const val);

// Load/Store
template <typename T> void Load(T &v, Scalar<T> const *ptr);
template <typename T> void Store(T const &v, Scalar<T> *ptr);

// Gather/Scatter
template <typename T, typename S = Scalar<T>>
T Gather(S const *ptr, Index<T> const &idx);

template <typename T, typename S = Scalar<T>>
void Scatter(T const &v, S *ptr, Index<T> const &idx);

// Masking/Blending
template <typename M> bool MaskFull(M const &mask);
template <typename M> bool MaskEmpty(M const &mask);

template <typename T>
void MaskedAssign(T &dst, const Mask<T> &mask, const T &src);

template <typename T>
T Blend(const Mask<T> &mask, const T &src1, const T &src2);

} // namespace vecCore
```

Listing 1: VecCore API

3. Mandelbrot Set with VecCore

A simple example that can be used to demonstrate how an algorithm changes when implemented using the VecCore API is the Mandelbrot set, shown in Figure 1 as calculated by our code. The Mandelbrot set can be obtained by repeatedly computing the function $f(z) = z^2 + c$ starting with $z = 0$ for various values of c in the complex plane. The final color depends on how many iterations of $f(z)$ are necessary for the point to leave the circle $|z| < 2$. Listings 2 and 3 show the scalar and vectorized implementations, respectively. The vectorized version works on several points at a time. Both algorithms are single-threaded for simplicity.

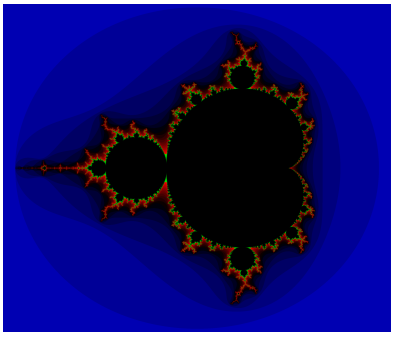

Figure 1. Mandelbrot Set

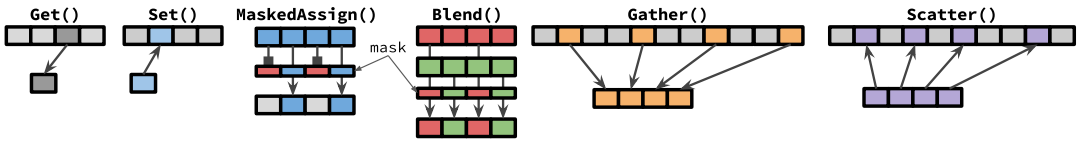

Figure 2. Illustration of VecCore API operations

```
template<typename T>
void mandelbrot(T xmin, T xmax, size_t nx,
                T ymin, T ymax, size_t ny,
                size_t max_iter, unsigned char *image)
{
    T dx = (xmax - xmin) / T(nx);
    T dy = (ymax - ymin) / T(ny);

    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; ++j) {
            size_t k = 0;
            T x = xmin + T(i) * dx, cr = x, zr = x;
            T y = ymin + T(j) * dy, ci = y, zi = y;

            do {
                x  = zr*zr - zi*zi + cr;
                y  = 2.0 * zr*zi + ci;
                zr = x;
                zi = y;
            } while (++k < max_iter && (zr*zr+zi*zi < 4.0));

            image[ny*i+j] = k;
        }
    }
}
```

Listing 2: Scalar implementation

```
template<typename T>
void mandelbrot_v(Scalar<T> xmin, Scalar<T> xmax, size_t nx,
                  Scalar<T> ymin, Scalar<T> ymax, size_t ny,
                  Scalar<Index<T>> max_iter,
                  unsigned char *image)
{
    T iota;
    for (size_t i = 0; i < VectorSize<T>(); ++i)
        Set<T>(iota, i, i);

    T dx = T(xmax - xmin) / T(nx);
    T dy = T(ymax - ymin) / T(ny), dyv = iota * dy;

    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; j += VectorSize<T>()) {
            Scalar<Index<T>> k{0};
            T x = xmin + T(i) * dx,       cr = x, zr = x;
            T y = ymin + T(j) * dy + dyv, ci = y, zi = y;

            Index<T> kv{0};
            Mask<T> m{true};

            do {
                x = zr*zr - zi*zi + cr;
                y = T(2.0) * zr*zi + ci;
                MaskedAssign<T>(zr, m, x);
                MaskedAssign<T>(zi, m, y);
                MaskedAssign<Index<T>>(kv, m, ++k);
                m = zr*zr + zi*zi < T(4.0);
            } while (k < max_iter && !MaskEmpty(m));

            for (size_t k = 0; k < VectorSize<T>(); ++k)
                image[ny*i+j+k] = (unsigned char) Get(kv, k);
        }
    }
}
```

Listing 3: VecCore implementation

The two implementations are quite similar to each other, the main differences can be seen inside the innermost `do/while` loop, that in the vectorized implementation must account for the differences when treating several points at the same time. The vectorized implementation requires using a mask to assign only to the right indices, and checking for termination becomes more complex due to that as well, since care must be taken to not return from the inner loop until all points in the SIMD vector have been properly computed.

		Intel Core i7 6700			**Intel Xeon Phi 7210**	
Runtime (ms)		GCC-7.2	Clang-5.0	ICC-18.0	GCC-7.2	ICC-18.0
Single	Scalar	550	549	677	3415	3609
Precision	Scalar Backend	570	569	677	3353	3510
	Vc 1.3.3	110	110	126	1064	1160
	UME::SIMD 0.8.1	117	117	131	–	543
Double	Scalar Algorithm	548	548	672	3409	3602
Precision	Scalar Backend	571	571	674	3348	3502
	Vc 1.3.3	267	267	257	2101	2087
	UME::SIMD 0.8.1	421	421	422	–	846
Speedup vs Scalar						
Single	Scalar Backend	0.96	0.96	1.00	1.02	1.03
Precision	Vc 1.3.3	5.00	4.99	5.37	3.21	3.11
	UME::SIMD 0.8.1	4.70	4.69	5.17	–	6.65
Double	Scalar Backend	0.96	0.96	0.99	1.02	1.03
Precision	Vc 1.3.3	2.05	2.05	2.61	1.72	1.73
	UME::SIMD 0.8.1	1.30	1.30	1.59	–	4.25

Table 1. Runtime in milliseconds of Mandelbrot set example for various configurations.

Runtime and speedups are shown in Table 1. It is important to note the difference in frequency between the Core i7 (3.4 GHz) and the Xeon Phi (1.3 GHz), hence the difference in absolute times. Moreover, performance comparisons using Julia sets (similar to Mandelbrot, but fixing c and varying z) show that the speedup strongly depends on branch divergence—that is, how many times the vectorized loop can return early. Complex regions of the fractal decrease the maximum speedup that can be obtained with vectorization. Therefore, these results should be used only as a relative measurement of performance between backends, and not as a measurement of the overall vectorization efficiency.

4. Electromagnetic Physics Models

Collision events at the LHC frequently produce energetic photons and electrons that lead to electromagnetic particle showers containing a large number of particles. Since particles generated in these showers constitute the vast majority of particles generated in a collision event, the simulation of electromagnetic processes is computationally expensive. Therefore, these processes are the natural first candidates for vectorization.

In the GeantV project [1, 2], VecCore has been used in the implementation of vectorized versions of electromagnetic physics models. Figure 3 shows the vectorization speedups for each model on an Intel Xeon Phi (Knights Corner) for the main electromagnetic processes. For 64 or more tracks, the speedup is between 5x and 6.5x in double precision, when the maximum speedup is 8x. Considering that these models rely on gather/scatter operations that are not so fast compared with other SIMD operations such as arithmerics, masking, etc, this result is very promising. Nevertheless, these speedups are expected to increase once a vectorized pseudo-random number generator is implemented in the framework.

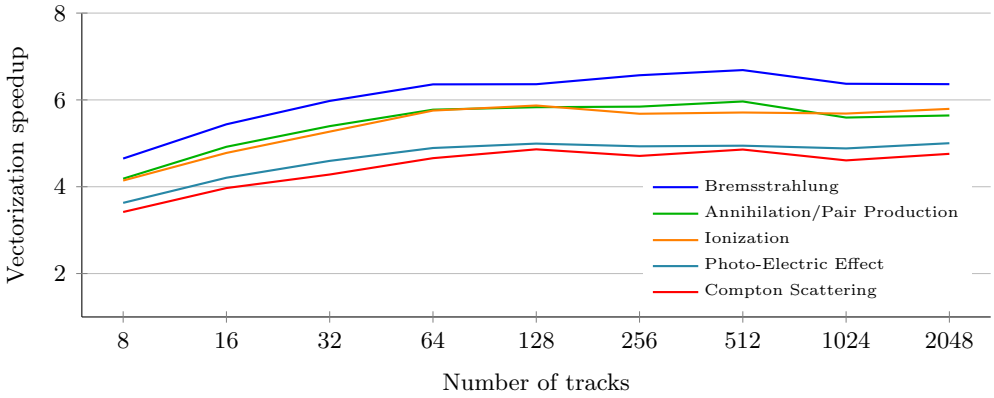

Figure 3. Vectorization speedup of electromagnetic models on Intel Xeon Phi

5. Geometry Algorithms

VecGeom[7, 8], the geometry library used in GeantV [1, 2, 9] and recent versions of Geant4 [10, 11, 12], has introduced a vectorized multi-particle API based on VecCore to perform ray casting, distance calculations, and navigation in sets of particles within detector geometries. Performance gains on Intel Knights Landing for VecGeom shapes compared to previous implementations are shown in Figure 4. Since Vc does not support KNL, the scalar backend has been used (i.e. speedup comes from compiler auto-vectorization only). As expected, in this case the UME::SIMD backend outperforms Vc by a significant margin.

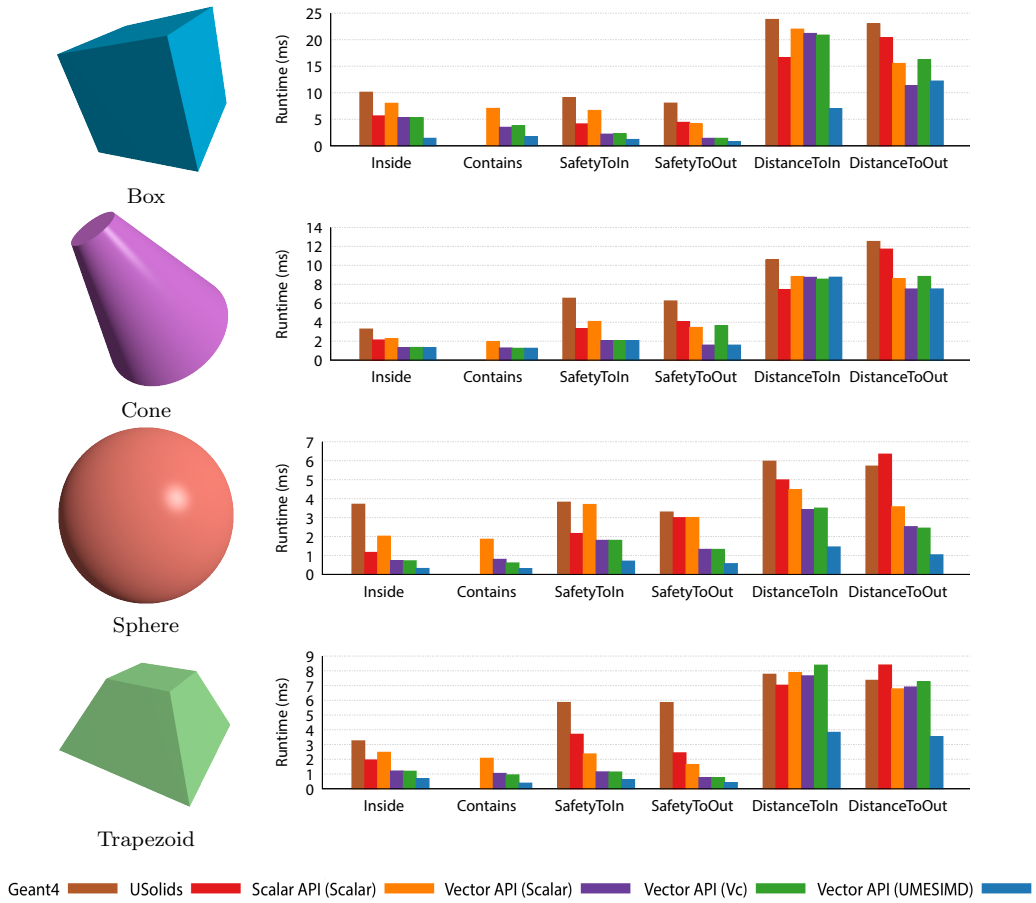

Figure 4. Vectorization speedups of selected shapes on Intel Knights Landing (KNL)

6. Conclusions

VecCore introduces a new unified abstraction layer on top of existing SIMD libraries that lets users write generic vectorized code that can be used with different instruction sets and hardware architectures. It also allows the user to change libraries to take advantage of the strengths of each library, or avoid drawbacks like lack of support for some architectures (e.g. KNL, ARM, or PowerPC), since VecCore has a simple scalar backend that works across architectures. If new backends are added, users of VecCore can take advantage of them without needing to change their code, by only changing the backend configuration. These features make VecCore a useful library for writing SIMD-enabled algorithms that are expected to perform well on a variety of architectures.

7. Acknowledgements

VecCore development has been funded by an Intel Parallel Computing Center project at São Paulo State University (UNESP), Fermilab, and CERN, as part of the GeantV project. VecCore is now maintained by the ROOT Team.

References

[1] Amadio G, Apostolakis J, Bandieramonte M, Bhattacharyya A, Bianchini C, Brun R, Canal P, Carminati F, Duhem L, Elvira D, de Fine Licht J, Gheata A, Iope R L, Lima G, Mohanty A, Nikitina T, Novak M, Pokorski W, Seghal R, Shadura O, Vallecorsa S and Wenzel S 2015 *Journal of Physics: Conference Series* **664** 072006 URL http://stacks.iop.org/1742-6596/664/i=7/a=072006

[2] Apostolakis J, Brun R, Carminati F and Gheata A 2012 *International Conference on Computing in High Energy and Nuclear Physics 2012 (CHEP2012)* **396** 022014 URL http://iopscience.iop.org/1742-6596/396/2/022014/pdf/1742-6596_396_2_022014.pdf

[3] ROOT Data Analysis Framework URL https://root.cern

[4] Kretz M and Lindenstruth V 2011 *Software: Practice and Experience* **42** 1409–1430 URL http://onlinelibrary.wiley.com/doi/10.1002/spe.1149/abstract

[5] Karpiński P and McDonald J 2017 A high-performance portable abstract interface for explicit SIMD vectorization *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores* PMAM'17 (New York, NY, USA: ACM) pp 21–28 ISBN 978-1-4503-4883-6 URL http://doi.acm.org/10.1145/3026937.3026939

[6] Vallecorsa S and Amadio G Auto-vectorization: recent progress *21st Geant4 Collaboration Meeting* (2016) URL https://agenda.infn.it/contributionDisplay.py?contribId=125&sessionId=22&confId=11196

[7] Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2014 *Journal of Physics: Conference Series* **513** 052038 URL http://stacks.iop.org/1742-6596/513/i=5/a=052038

[8] Apostolakis J, Bandieramonte M, Bitzes G, Brun R, Canal P, Carminati F, Cosmo G, Licht J C D F, Duhem L, Elvira V D, Gheata A, Jun S Y, Lima G, Nikitina T, Novak M, Sehgal R, Shadura O and Wenzel S 2015 *Journal of Physics: Conference Series* **608** 012023 URL http://stacks.iop.org/1742-6596/608/i=1/a=012023

[9] Apostolakis J, Brun R, Carminati F, Gheata A and Wenzel S 2014 *Journal of Physics: Conference Series* **523** 012004 URL http://stacks.iop.org/1742-6596/523/i=1/a=012004

[10] et al S A 2003 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **506** 250 – 303 ISSN 0168-9002 URL http://www.sciencedirect.com/science/article/pii/S0168900203013688

[11] Asai M 2012 *International Conference on Computing in High Energy and Nuclear Physics 2012 (CHEP2012)* **396** 052007 URL http://iopscience.iop.org/1742-6596/396/5/052007/pdf/1742-6596_396_5_052007.pdf

[12] et al J A 2006 *Nuclear Science, IEEE Transactions on* **53** 270–278 ISSN 0018-9499 URL http://dx.doi.org/10.1109/TNS.2006.869826